Representation and Manipulation of Inductive Boolean Functions

Aarti Gupta Allan L. Fisher

April, 1992
CMU-CS-92-129

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

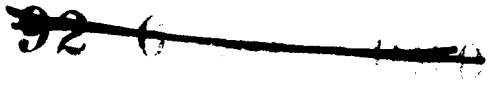
Abstract

The high level of complexity of current hardware systems has led to an interest in formal methods for proving their correctness. Reasoning by induction is a powerful formal proof method with the advantages that a single proof establishes the correctness of an entire family of circuits varying according to some size parameter, and the size of this proof is independent of the size of the circuit. Tautology-checking of symbolic Boolean functions is another method that has been successfully used in formal verification, both for checking equivalence/satisfaction with respect to a functional specification and as a basis for model-checking. In this paper we present a representation and algorithms for symbolic manipulation of inductive Boolean functions, based on extensions of Binary Decision Diagrams. Our approach allows us to combine reasoning by induction with efficient tautology-checking in an automatic way.

92

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.



Keywords: Formal Hardware Verification, Induction, Inductive Boolean Functions, Binary Decision Diagrams.

1. Introduction

Advances in hardware technology have led to an increased level of complexity in current hardware systems. Late detection of design errors typically results in higher costs due to the associated time delay as well as potential loss of production. It is therefore important that hardware designs be free of errors. Formal verification has become an increasingly promising technique towards establishing the correctness of hardware designs, as the traditional techniques like simulation are proving to be inadequate due to computational demands of the task involved.

Reasoning by induction is a powerful proof method for formal verification of hardware. Typically, the major units of a hardware system are described inductively in terms of size parameters. The main advantage of such an approach is that a single proof by induction (on a size parameter) establishes the correctness of an entire family of circuits. Moreover, the size of this proof is independent of the circuit size, and can therefore be used to reason about particular-sized instances of circuits also. It has been successfully employed by several research groups, both in a theorem-proving context [11, 16, 17], and within a model-checking/language-containment paradigm [12, 18, 20]. In the former set of approaches, it is typically used for verifying functional properties of parameterized designs. The reasoning process is partially automated in the sense that heuristics can be incorporated into a semi-automatic theorem-prover. However, a complete proof usually requires interaction with the user in order to set up useful intermediate lemmas and to guide the overall proof strategy. In the latter set of approaches, induction is typically used to verify behavioral properties of an unbounded number of identical processes. The invariant used for induction is provided by the user, though automated help is available to check the proofs. Thus, in all available approaches, reasoning by induction is a fairly complex process with semi-automated proofs at best.

Tautology-checking of symbolic Boolean formulas has also become prominent in the area of formal hardware verification, largely due to availability of symbolic manipulation algorithms that are efficient in practice [5, 6]. It has been used for functional verification of combinational circuits [6] and for Hoare-style verification of finite state sequential circuits [1, 4, 7], with respect to specifications expressed as symbolic Boolean formulas. By symbolically encoding the description of a finite state system, both the finite state machine equivalence problem and the model-checking problem (for various logics) have also been converted to tautology-checking problems on symbolic Boolean domains [3, 8, 9, 10, 13, 14, 15, 19]. The main advantage of these approaches, over traditional state enumeration approaches, is that they avoid representation of an explicit state-transition graph. This helps in handling large systems and the associated state explosion problem. However, these tautology-checking methods have so far been used to verify only fixed-sized instances of circuits, which could otherwise be defined parametrically in terms of size parameters.

The motivation for the research presented in this paper is to combine reasoning by induction and tautology-checking in a way that incorporates the advantages of both. On the one hand this would allow us to reason inductively about all instances (sizes) of a circuit described parametrically, and on the other this process would be completely automated and potentially efficient in practice. Moreover, it is frequently the case that induction is successfully used to prove the *individual* correctness of hardware units, while no mechanism is provided to reason about a *composition* of inductively-defined units. Our approach tackles the problem of composition also.

1.1. Overview of Our Verification Approach

The underlying basis of our approach consists of identifying certain classes of inductive Boolean functions[1] and representing them in a canonical form using extensions of Bryant's Binary Decision Diagrams (BDDs) [6]. A proof by induction for such a function can then be replaced by tautology-checking, which is completely automated with the help of canonical representations. We provide different schemata for different classes of functions, each schema consisting of a canonical representation (for a function in that class) and the associated symbolic manipulation algorithms. This approach is extensible in the sense that new classes of functions can be identified, and new schemata added, as and when needed.

Its application to formal hardware verification lies in appropriate representation of specifications and hardware in the form of inductive Boolean functions that can be handled by our schemata. We have already identified and explored some important classes of inductive Boolean functions that allow us to represent and verify a fair number of practical circuits. By supporting a composition semantics on these Boolean functions, we have also ensured support for induction proofs on compositions of inductively-defined hardware units.

In this paper, we concentrate on our underlying schema for a class of Boolean functions we call linearly inductive functions. Many of the ideas and algorithms described for this class can be generalized to apply to other classes of inductive Boolean functions as well. The details of our schemata for the other classes that we have identified, and the application of our methodology to formal hardware verification problems, will appear in a forthcoming paper.

1.2. Organization of the Paper

We start by providing a summary of Bryant's BDDs, which are used for canonical representation of arbitrary Boolean formulas [5, 6]. This is followed by our characterization of linearly inductive functions. After that we describe the progression of representations from a standard BDD to a Layered BDD (LBDD); the latter providing a motivation for the exact form of a Linear Inductive BDD (LIBDD), which we use to represent linearly inductive functions in a canonical form. We describe details of our algorithms for building and maintaining canonical LIBDD structures, and for symbolically manipulating linearly inductive functions, along with examples. This is followed by a complexity analysis and a comparison with standard BDD manipulations. We conclude by highlighting the aspects of our method that help in automation of induction proofs for linearly inductive functions.

2. Binary Decision Diagrams

We present a brief summary from Bryant's original paper on BDDs [6], borrowing terminology from a later paper [5] (without going into implementation details).

A Binary Decision Diagram is a directed acyclic graph (dag) on nodes that represent Boolean variables. Each such node has two outgoing edges corresponding to the variable being true (labeled '1') or false (labeled '0') respectively. There are two special nodes without any outgoing edges that denote the Boolean constants '1' and '0'. A function can be associated with each node of a dag, such that a directed path from that node to a Boolean constant node corresponds to an assignment of that Boolean constant value to the

[1] Boolean functions defined inductively; definitions are provided for each class that we identify.

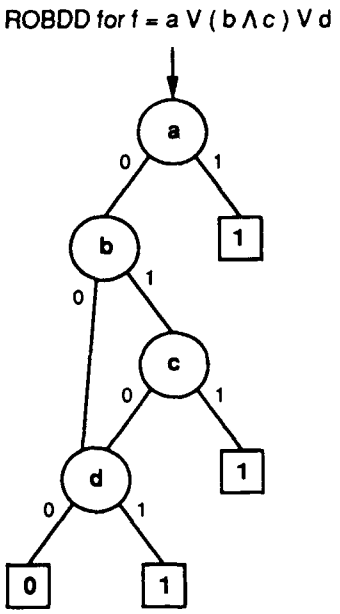


Figure 1: An Example ROBDD

function, for an assignment of variables as chosen by each edge along that path.

An Ordered BDD is a BDD with the additional constraint that all directed paths visit the variable nodes in a fixed ascending order, specified as a total ordering on variables for representation of all functions in the system. In general, the size of a BDD representation for a function is sensitive to this variable ordering.

A Reduced Ordered BDD (ROBDD) has the additional properties that there are no isomorphic sub-graphs or redundant variable nodes (both of whose edges point to the same sub-graph) in a BDD dag. For example, the ROBDD for a function $f = a \vee (b \wedge c) \vee d$ is shown in Figure 1 for the variable ordering $a < b < c < d$. (In the figure, edges implicitly point from top to bottom, variables nodes are shown as circles and special Boolean constant nodes are shown as boxes.)

Bryant has shown that an arbitrary Boolean formula can be represented by a unique (up to isomorphism) ROBDD [6]. Thus, tautology-checking of a Boolean formula, represented in a canonical form as an ROBDD, is equivalent to checking if its ROBDD is isomorphic to the Boolean constant node '1'. Moreover, equivalence of two functions can be checked simply by checking that their ROBDDs are isomorphic. In a typical implementation, an ROBDD is stored in a strong canonical form, thereby making the comparison check a constant-time operation [5].

Bryant also provides efficient graph algorithms for construction of ROBDDs and for symbolic manipulation of Boolean formulas, including Boolean operations for conjunction, disjunction and negation, and the operations for restriction and composition [6]. The complexity of these algorithms is typically linear in the size of the ROBDDs of the corresponding functions. Though the size of an ROBDD for a function can be exponential in the worst-case, many functions encountered in typical applications have reasonable representations, thereby leading to efficient symbolic Boolean manipulation in practice.

3. Linearly Inductive Functions

Informally, an inductive (Boolean) function is described in a parametric form using an induction parameter, with the arguments (inputs) of the function also parameterized accordingly. We characterize a linearly inductive function f as satisfying the following three conditions :

Condition 1: For $i = 1$, the 1-instance of f (denoted f^1) is a Boolean combination (denoted $\mathcal{B}^1$) of the 1-instance inputs (denoted X^1),
i.e. $f^1 = \mathcal{B}^1(X^1)$.

Condition 2: For all $i > 1$, the i-instance of f (denoted f^i) is a Boolean combination (denoted $\mathcal{B}^i$) of only the i-instance inputs (denoted X^i) and some $(i-1)$-instance functions (denoted G^{i-1}), each of which also satisfies Conditions 1, 2 and 3,
i.e. $f^i = \mathcal{B}^i(X^i, G^{i-1})$.

Condition 3: For all $i, j > 1, \mathcal{B}^i = \mathcal{B}^j$,
i.e. f^i is related to its inputs (X^i and G^{i-1}) in the same manner as f^j is to its inputs (X^j and G^{j-1}).

4. Description of a Layered BDD (LBDD)

An LBDD is best thought of as a layered version of a BDD, with an associated parameter called *level*. The variables of this BDD-like representation are also parameterized accordingly, and a parameterized variable instantiated with parameter value i is called an i-level variable. An i-level LBDD is described inductively as follows :

1. A 1-level LBDD is a BDD with 1-level variables.
2. For $i > 1$, an i-level LBDD (shown in Figure 2) is like a BDD with i-level variables, except that its leaf nodes can be :
   - Boolean constants, or
   - pointers to other $(i-1)$-level LBDDs.

By imposing an ordering on the parameterized variables we obtain Ordered LBDDs. Note that we specify an ordering on the parameterized variables, not their i-level instances. This ordering is implicitly applied for all $i \geq 1$, to obtain the ordering for the i-level variables. For example, if an ordering $x < y < z$ is specified for parameterized variables x, y and z; it implies that for all $i \geq 1$, i-level instance of $x < i$-level instance of $y < i$-level instance of z.

A function f_v is associated with the root v of an i-level Ordered LBDD and is recursively defined in the same manner as for a BDD, except when v is a leaf and points to an $(i-1)$-level LBDD with root vertex w. In this case, f_v is equal to f_w, the latter being well-defined by an inductive argument on the basis case for $i = 1$ (standard BDD) and the fact that f_w is independent of the i-level variables.

We also extend the BDD operation *Reduce* [6] to an LBDD-Reduce operation on an i-level Ordered LBDD. This operation is similar to that for a BDD, with the modification that leaf nodes are taken to be isomorphic if and only if :

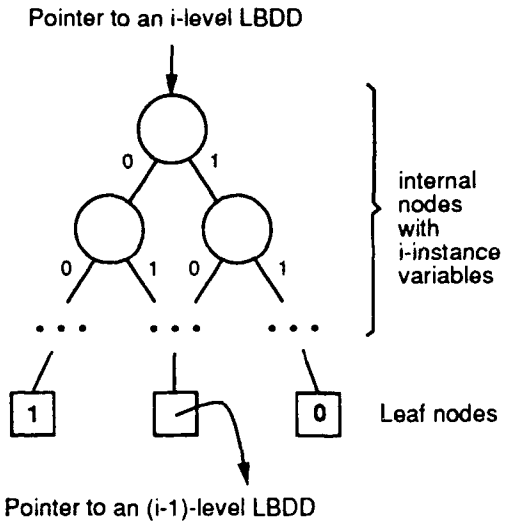


Figure 2: An i-Level LBDD

- they denote the same Boolean constant, or
- they point to the same $(i-1)$-level Reduced Ordered LBDD.

The resulting LBDD is called a Reduced Ordered LBDD.

4.1. Canonicity Property of an LBDD

Theorem 1 : For a given ordering of the parameterized inputs, for all $i \geq 1$, f^i for a linearly inductive function f can be represented in a canonical form using an i-level Reduced Ordered LBDD.

Proof : By induction on i.

Basis Case : $i = 1$.
It is easy to see that f^1 can be represented by an ROBDD over variables representing X^1. This ROBDD is a 1-level Reduced Ordered LBDD by definition, and is canonical due to the canonicity property of an ROBDD.

Induction Hypothesis : Let the theorem be true for $i = k$.

Induction Step : Consider f^{k+1}. It is a Boolean combination of X^{k+1} and G^k. Note from Condition 2, that Boolean combinations of k-instance functions are themselves k-instance functions. By the induction hypothesis, each of these can be represented in a canonical form as a k-level Reduced Ordered LBDD. Therefore f^{k+1} can be represented as an Ordered BDD with instantiated parameterized variables representing X^{k+1} as non-leaf nodes, and with leaf nodes that potentially point to k-level Reduced Ordered LBDDs. In other words, f^{k+1} can be represented as a $(k+1)$-level Ordered LBDD.

We next apply the LBDD-Reduce operation to this $(k+1)$-level Ordered LBDD. The leaf node comparison in this operation is straightforward since the leaves are either Boolean constants, or they point to k-level Reduced Ordered LBDDs, which are canonical by the induction hypothesis. The rest of the LBDD-Reduce

operation is similar to that for an Ordered BDD with variables representing X^{k+1}. Thus, the resulting Reduced Ordered $(k+1)$-level LBDD is canonical for f^{k+1}, due to canonicity of the leaf nodes and canonicity of an ROBDD on variables representing X^{k+1}.

□

To recapitulate, we have established the correspondence between f^i and (the function associated with the root of) an i-level LBDD. Just as f^i is defined in terms of i-instance inputs and $(i-1)$-instance functions, the corresponding i-level LBDD is defined in terms of i-level variables and pointers to $(i-1)$-level LBDDs. For the rest of this paper, we assume that BDDs and LBDDs are Reduced and Ordered, unless otherwise stated.

4.2. Relationship between a BDD and an LBDD

A natural question at this point concerns the correspondence, if any, between an i-level LBDD representation of f^i and a simple BDD representation of f^i. We first establish a correspondence between the variable orderings used by each. Recall that an i-level Ordered LBDD uses an ordering on the parameterized variables which is reflected in the ordering between their i-level instances for all $i \geq 1$. However, a total ordering for a BDD for f^i has to specify an ordering between the j-level and the $(j-1)$-level variables also (for $1 < j \leq i$). We therefore define the notion of a *consistent* ordering. An ordering σ, on all i-level instances of all parameterized variables, is consistent with an ordering ρ on parameterized variables, if :

- for all $i \geq 1$, σ preserves the relationship between the i-level instances of the parameterized variables according to that specified in ρ, and
- for all $i > 1$, i-level variables $<$ $(i-1)$-level variables in σ.

Note that if f is regarded as a multiple-output function (each output being $f^i, i \geq 1$), then σ captures the natural variable ordering for a BDD representation of f, which consists of a single graph with multiple roots corresponding to multiple outputs [6].

It is now easy to see that LBDD representations (for a particular ordering ρ) are a way of modularizing monolithic BDD representations (for an ordering σ consistent with ρ) into layers, as demonstrated by the following example.

Example 1 : Let $f^i = x^i \wedge \neg y^i \wedge f^{i-1}$ for $i > 1$, and let $f^1 = x^1 \wedge \neg y^1$, where
x^i and y^i represent the i-instances of parameterized inputs x and y.

Let the LBDD ordering be $\rho : x < y$.
Consider a BDD representation of f^i with the variable ordering
$\sigma : x^i < y^i < x^{i-1} < y^{i-1} \ldots x^1 < y^1$ (which is consistent with ρ).

Imagine partitioning this BDD into i number of layers, such that each layer j consists of only the section of the BDD dealing with variables x^j and y^j, as shown in Figure 3. A pointer into a layer j represents the contribution of the j-level LBDD to the $(j+1)$-level LBDD, with the bottom-most layer (layer 1) having pointers to only Boolean constants. Each j-level LBDD effectively captures layer j of the BDD, thereby modularizing the BDD representation.

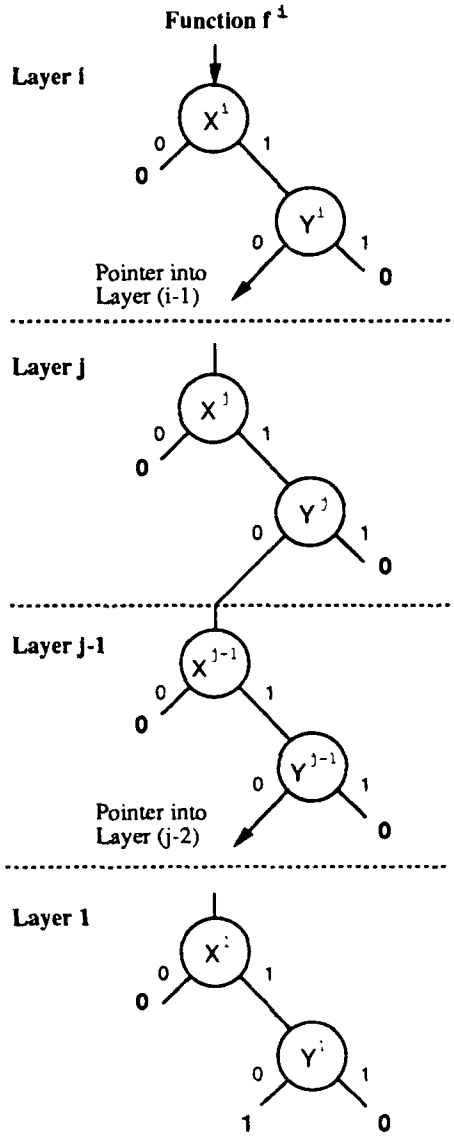


Figure 3: Partition of a BDD into Layers

5. Description of a Linear Inductive BDD (LIBDD)

A naive representation of a system of linearly inductive functions could maintain explicit structures for all i-level LBDDs of interest, corresponding to f^i for each function f. However, this would not result in any space savings in comparison to representing each f^i by a standard BDD. The reason is that in order to maintain an i-level LBDD for f^i, one would also have to maintain all j-level LBDDs, $1 \leq j < i$, that this i-level LBDD could recursively point to. Moreover, this would allow representations for only bounded values of i, not for *all* $i \geq 1$.

The real advantage of the LBDD representation for linearly inductive functions lies in exploiting Condition 3. For all $i, j > 1$, $\mathcal{B}^i = \mathcal{B}^j$ implies that the graph structure of the i-level LBDD representation for f^i is isomorphic to the graph structure of the j-level LBDD representation for f^j, *modulo* substitution of i-level instances of parameterized variables X for corresponding j-level instances at the non-leaf nodes, and substitution of $(i-1)$-level LBDDs for G for corresponding $(j-1)$-level LBDDs at the leaf nodes. In other words, the graph structure for all i-level LBDDs, $i > 1$, can be captured by maintaining a substitution semantics on only one parametric LBDD structure.

Thus, a linearly inductive function f can be identified with a *function descriptor* consisting of a pair of pointers, one to each of the following two structures :

1. a 1-level LBDD representing the 1-instance of f, called the *Basis BDD* for f.
2. a parametric k-level LBDD with substitution semantics, representing all i-instances of f, $i > 1$, called the *Linear Inductive BDD (LIBDD)* for f.

The intended semantics of the substitutions is such that pointers to a 1-level LBDD representing f^1 are taken to be pointers to the Basis BDD for f. Pointers to i-level LBDDs representing f^i, $i > 1$, are taken to be pointers to the LIBDD for f *with an appropriate substitution of* i *for* k.

Let us consider the substitution semantics in a little more detail. Recall that a k-level LBDD representing f^k has leaf nodes that (potentially) point to $(k-1)$-level LBDDs representing G^{k-1}. According to our substitution semantics, for $i > 2$, the leaf nodes of an i-level LBDD for f^i should point to the LIBDDs for G, with the substitution $(i-1)$ for k. However, for $i = 2$, the leaf nodes of a 2-level LBDD for f^2 should point to the Basis BDDs, and not to the LIBDDs, for G. Thus, the leaf nodes of a parametric k-level LBDD for f^k, admitting all substitutions $i > 1$ for k, need to point to the *function descriptors* (Basis BDDs and LIBDDs) for G. To summarize, an LIBDD is represented as a parametric k-level LBDD with leaf nodes that (potentially) point to function descriptors (along with the substitution $(k-1)$ for k).

It is clear that canonicity of the Basis BDDs for all linearly inductive functions can be maintained by any of the schemes used in a generic BDD package [5]. However, we still need to address the issue of canonicity of an LIBDD representation for all $f^i, i > 1$. (Note that Theorem 1 in Section 4.1 talks only about canonicity of an i-level LBDD representation for f^i, not about an LIBDD representation with substitution semantics for all $f^i, i > 1$.) We address this important issue in the next section, as it forms the core of our approach for reasoning about linearly inductive functions.

6. Building and Maintaining Canonical LIBDDs

For this section, consider all functions to be linearly inductive. As described in the previous section, an LIBDD for f is like a k-level LBDD for f^k, except that its leaf nodes point to function descriptors for G instead of pointing to $(k-1)$-level LBDDs for G^{k-1}. Recall from the proof of Theorem 1 in Section 4.1, that the canonicity argument for a k-level LBDD representing f^k depends upon performing an LBDD-Reduce operation after it is ascertained that the leaf nodes point to canonical structures. The same argument can be applied to an LIBDD for f. Its canonicity is also maintained by performing an LBDD-Reduce operation after establishing the canonicity of structures pointed to by its leaf nodes.

There are two main problems that need to be addressed in order to establish canonicity of leaf nodes of an LIBDD.

1. A set of functions may be described in a mutually-recursive manner. In this case, a bottom-up approach starting from canonical leaf nodes, as is used in building a standard BDD, will not work.
2. The definition of an i-instance function may involve arbitrary Boolean combinations of $(i-1)$-instance functions. This can be handled by renaming a Boolean combination of two given functions as a new function. However, we have to guard against creation of an infinite series of new functions.

6.1. Tentative Function Descriptors

We handle the first of these problems by working initially with "tentative" function descriptors, one for each user-specified function. The tentative Basis BDDs and the tentative LIBDDs mirror the definitions provided by the user, with an implicit understanding that the leaf nodes of a tentative LIBDD may point to non-canonical function descriptors. (The Basis BDDs are canonical by construction.)

The second problem is handled by maintaining canonical forms for Boolean combinations *with respect to user-specified functions*, i.e. canonicity is maintained with respect to user-specified functions, not with respect to the underlying variables. For example, two combinations p and q, defined to be $f \vee g$ and $f \wedge g$ respectively, are not equivalent with respect to functions f and g. However, they may be equivalent with respect to the underlying variables x and y, e.g. when f and g represent equivalent functions over x and y. The task of maintaining canonicity with respect to user-specified functions is easily accomplished by working in a separate Boolean meta-space, where a meta-variable is assigned to each user-specified function. BDDs on these meta-variables, called Meta-BDDs, denote arbitrary Boolean combinations of these functions. Due to the canonicity property, Meta-BDDs ensure that all equivalent combinations are renamed to the same new function. Since there are a finite number of unique combinations for a finite number of user-specified functions, this also guarantees termination of the renaming process.

For each new function created to represent a Boolean combination, we create a tentative function descriptor as follows. Its Basis BDD is formed by performing a standard BDD *Apply* operation [6], corresponding to the Boolean combination operator, on the Basis BDDs for the argument functions. Its tentative LIBDD is constructed by using a modified Apply operation on the tentative LIBDDs for the argument functions. The simple modification consists of terminating recursion at the leaf nodes by creating new functions representing distinct Boolean combinations, as and when needed.

In summary, a tentative function descriptor is generated for each user-specified function, and for each distinct Boolean combination (of user-specified functions) that is needed. This process is illustrated by the

following example.

Example 2 : Consider the following description of mutually-recursive user-specified functions f, g and h.

for all $i > 1, f^i = x^i \wedge g^{i-1}$ and $f^1 = x^1$,
for all $i > 1, g^i = x^i \wedge h^{i-1}$ and $g^1 = x^1$,
for all $i > 1, h^i = x^i \wedge (f^{i-1} \vee g^{i-1})$ and $h^1 = x^1$.

The tentative function descriptors (TFDs) for functions f, g and h are shown in Figure 4(a). (In our figures, the first slot of a TFD is understood to contain a pointer to the corresponding Basis BDD. We do not show these pointers in order to avoid clutter.) In the same figure, we have also shown tentative function descriptors for new functions p, q and r which represent the required Boolean combinations ($p^i = f^i \vee g^i$, $q^i = g^i \vee h^i$ and $r^i = h^i \vee p^i$). Note that the new functions — p, q and r — represent distinct Meta-BDDs consisting of the meta-variables f, g and h as shown in Figure 4(b). At this stage, the tentative LIBDD for r has a leaf node that points to a potentially new function r' denoting the combination $(p^i \vee q^i)$. From manipulations in the meta-space, we find that r' represents the same meta-function as r. Thus, the leaf node of the LIBDD for r should point back to itself (with the substitution $(k-1)$ for k). If we had not caught this redundancy, the process of renaming combinations to new functions would have continued forever. Thus, capturing the canonicity of Boolean combinations (with respect to user-specified functions) is essential for termination of the renaming process.

6.2. Making Tentative Function Descriptors Canonical

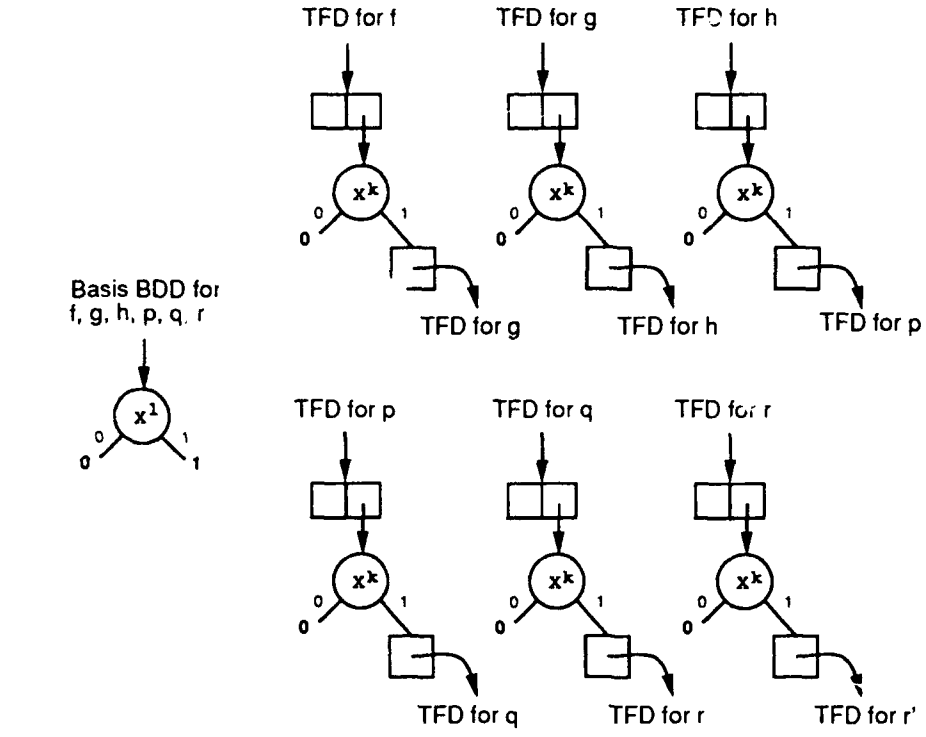
Once all required, potentially mutually-defined, tentative function descriptors have been obtained, the task is to convert them to canonical function descriptors. We maintain a separate set of canonical function descriptors (which is empty to start with) in a manner similar to a standard scheme for BDDs [5]. The idea is to convert each tentative function descriptor to a canonical one, by either finding one in the existing canonical set that it is equal to, or by adding a new member to this set if it is unequal to all existing members. A high-level algorithm to check equality of a tentative function descriptor against another function descriptor (tentative or canonical) is shown in Figure 5 as Algorithm A.

Since the leaf nodes of a tentative LIBDD may point to non-canonical function descriptors, Step 4 of Algorithm A may recursively lead to new equality checks on pairs of function descriptors. The termination condition for this recursion involves an interesting interplay between the mutual-recursive and inductive aspects of function definitions. It is based on the observation that checking equality of two function descriptors for f and g, is equivalent to checking if $f^i = g^i$ by induction on i, for all $i \geq 1$. This is best illustrated with the help of an example.

Example 3 : Consider a set of user-specified functions f, g, h, p, q and r, defined as follows :

for $i > 1, f^i = (\neg x^i \wedge g^{i-1}) \vee (x^i \wedge h^{i-1})$ and $f^1 = x^1$,
for $i > 1, g^i = (\neg x^i \wedge p^{i-1}) \vee (x^i \wedge q^{i-1})$ and $g^1 = x^1$,
for $i > 1, h^i = (\neg x^i \wedge p^{i-1}) \vee (x^i \wedge f^{i-1})$ and $h^1 = x^1$,
for $i > 1, p^i = (\neg x^i \wedge g^{i-1}) \vee (x^i \wedge r^{i-1})$ and $p^1 = x^1$,
for $i > 1, q^i = (\neg y^i \wedge f^{i-1}) \vee (x^i \wedge g^{i-1})$ and $q^1 = y^1$,
for $i > 1, r^i = (\neg y^i \wedge h^{i-1}) \vee (x^i \wedge p^{i-1})$ and $r^1 = y^1$.

The tentative function descriptors for these functions are mutually-recursive, and simply mirror the definitions above. Assume that our set of canonical function descriptors is empty (as is the case initially). So we



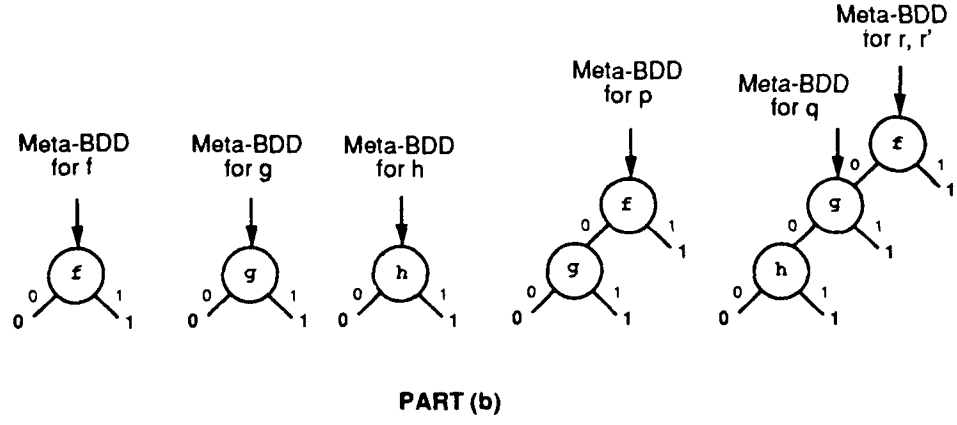


Figure 4: TFDs and Meta-BDDs for Example 2

Checking equality of a function descriptor s against another function descriptor t :

1. Check equality of Basis BDDs of s and t.
   If not equal, $s \neq t$.
2. Perform an LBDD-Reduce operation on LIBDD for s.
3. Check if graph of LIBDD for s on k-instance variables is isomorphic to that for t, modulo equality of corresponding leaf nodes.
   If not isomorphic, $s \neq t$.
4. Check if all pairs of corresponding leaf nodes of the two LIBDDs are equal.
   If not equal, $s \neq t$.
   Otherwise $s = t$.

Figure 5: Algorithm A

pick a candidate, say f, and attempt to make it canonical. The left leaf of the LIBDD for f (corresponding to $\neg x^k$) points to the tentative function descriptor for g. We therefore need to make the descriptor for g canonical. Since we had already decided to include f in our canonical set, we first need to check if g is equal to f. We use our Algorithm A on the pair (g, f). Steps 1 and 3 of this algorithm succeed. Step 4, however, recursively leads to equality checks for the pairs (p, g) and (q, h). In other words, the situation thus far is that $g = f$, if $p = g$ and $q = h$.

In general, we represent such dependencies conveniently in the form of a directed graph on nodes that represent equalities between (unordered) pairs of function descriptors. The intended semantics of this graph, which we call a Comparison-Graph, is that an edge exists from a node v to a node w, if the equality of (the pair of descriptors corresponding to) v depends upon the equality of (the pair of descriptors corresponding to) w. More formally, an edge from v to w exists if non-equality of w implies non-equality of v. Conversely, equality of a node v holds if equalities hold for all nodes pointed to by edges from v. We maintain a separate table of known equality results (equal/unequal) between pairs of descriptors. This table initially contains entries for pairs with identical elements (indicating their equality) and gets updated as more results are obtained.

The recursive application of Algorithm A is accompanied by the construction of a Comparison-Graph for each top-level call as follows. Starting with a node representing the top-level pair of descriptors, new nodes are recursively explored in a depth-first manner. Exploration of a node consists of the following case analysis :

Case 1 : If the descriptors in the pair are known to be equal (from the table computed thus far), we label the node 'True'.

Case 2 : If they are known to be unequal, we label the node 'False'.

Case 3 : Otherwise, we apply Algorithm A to the pair of descriptors.

- **(a)** : If either Step 1 or Step 3 of Algorithm A fails, we label the node 'False'.
- **(b)** : Otherwise, we have to construct edges from this node to all nodes that the equality of this node depends upon, as determined by Step 4 of Algorithm A. If nodes to be pointed to are not already present in the graph, new nodes are created and explored recursively in a depth-first manner.

Exploration of a node is complete after all required edges out of it have been constructed (along with exploration of newly created nodes).

A top-level call to Algorithm A (and the associated graph construction) terminates either as soon as we encounter a 'False' node (Cases 2 or 3(a) above) or after exploration of all nodes in the graph is complete. Note that due to a depth-first strategy for node creation and exploration, upon termination due to a 'False' node all incompletely explored nodes lie on a directed path from the top-level node to the 'False' node. For Example 3, we obtain the Comparison-Graph as shown in Figure 6(a).

We now prove the following assertion regarding a Comparison-Graph.

Theorem 2 : The equality of descriptors corresponding to a node v in a Comparison-Graph is true if and only if there is no directed path from v to a 'False' node.

Proof : Assume there is a directed path from v to a 'False' node. Note that a node labeled 'False' denotes a pair of descriptors that are unequal, either due to the result of a previous computation (Case 2) or due to non-equality that can be immediately determined from the application of Algorithm A (Case 3(a)). According to the semantics of our Comparison-Graph, a directed path from v to a 'False' node represents a chain of dependencies which ends in a pair of unequal descriptors. By transitivity of the implication relation, the descriptors corresponding to v are also unequal. (In fact, the length i of the shortest path to a 'False' node can be used to provide a particular instance of the respective functions which demonstrates the non-equality, thereby providing a counter-example facility.)

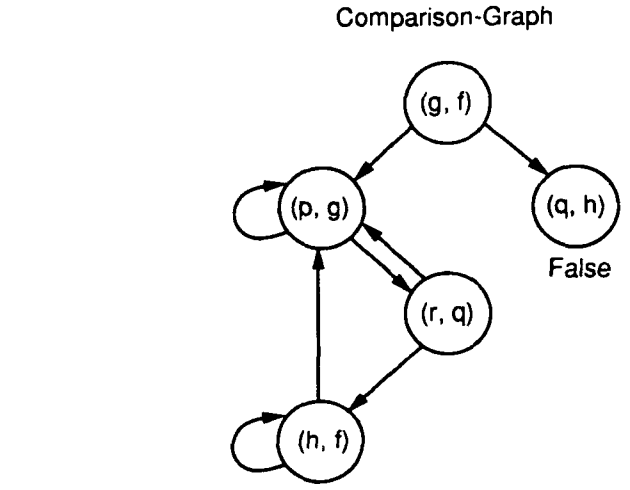
For a proof for the other direction, assume there is no directed path from v to a 'False' node. In order to prove that the equality of descriptors corresponding to v is true, we need to prove that *all* dependencies for v are satisfied, i.e. they allow $f^i = g^i$ for $i \geq 1$, where v represents the descriptor pair (f, g). We prove this by showing that each dependency chain, corresponding to a directed path from v, is satisfied. Note that since a 'False' node is unreachable from v, all nodes along all directed paths from v are completely explored, i.e. all required dependency edges have been constructed. Thus, examining all directed paths from v ensures that all dependencies are examined.

Now, each directed path from v satisfies one of the following conditions :

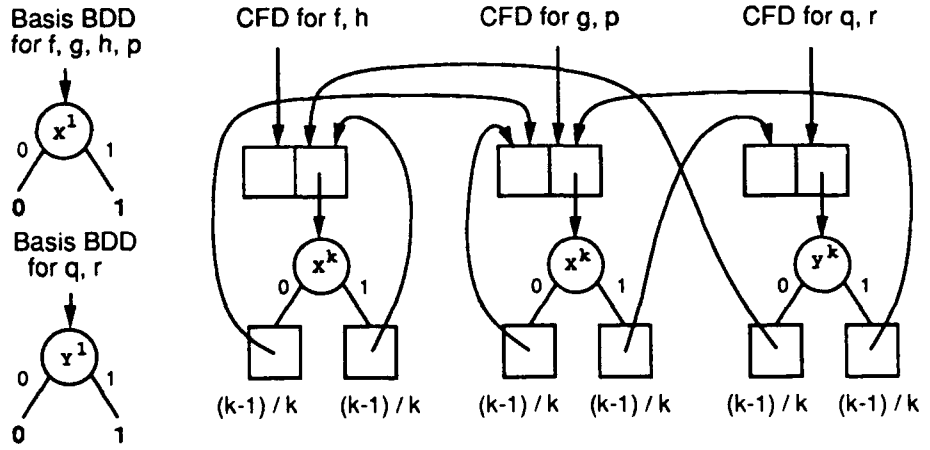
- it ends in a node labeled 'True', or
- it connects v to a cycle.

We show that the dependency chain along a directed path is satisfied under both conditions. As before, let node v represent the descriptor pair (f, g). Also, let the node labeled 'True' represent the descriptor pair (p, q) in the first condition, and let the first cycle node on the directed path represent the descriptor pair (h, l) in the second condition, as shown in Figure 7 (parts (a) and (b) respectively).

For the first condition, let length of the directed path to the 'True' node be j ($j \geq 1$). Note that the node labeled 'True' indicates $p^i = q^i$ for all $i \geq 1$, known due to result of a previous computation. Recall that Steps 3 and 4 of the top-level call to Algorithm A check the equality of two k-level LBDDs (LIBDDs), and that each recursive call to Algorithm A *implicitly* lowers the level parameter of the LBDDs being checked. Using this fact, and the dependency semantics of our Comparison-Graph, we can immediately infer that this path allows $f^i = g^i$ for all $i > j$. Moreover, for $1 \leq i \leq j$, the equality $f^i = g^i$ is allowed due to equality of the Basis BDDs (checked by Step 1 of Algorithm A) for the $(j + 1)$ pairs represented along the path (including the pair (f, g)). Thus, this path allows $f^i = g^i$ for all $i \geq 1$.



PART (a)



PART (b)

Figure 6: Comparison-Graph and CFDs for Example 3

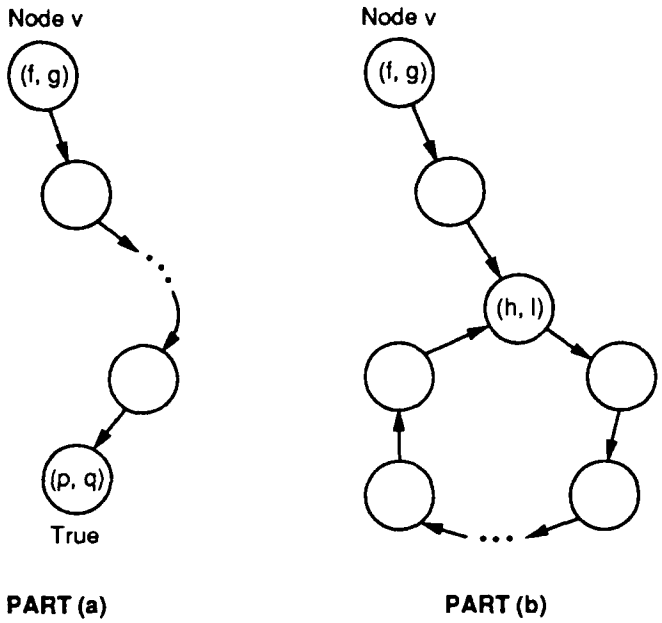


Figure 7: Directed Paths in a Comparison-Graph

For the second condition, let length of the non-cycle path be j ($j \geq 0$), and let length of the cycle be c ($c \geq 1$). First consider the cycle node representing the pair (h, l). Step 1 of Algorithm A ensures equality of the Basis BDDs for h and l, i.e. $h^1 = l^1$. By going around the cycle once, we obtain a dependency chain such that equality of h^k and l^k depends upon the equality of h^{k-c} and l^{k-c}. By an inductive argument on i, we can infer that this cycle allows $h^i = l^i$ for all $i \geq 1$. Using this cycle node in place of a 'True' node, we use an argument similar to the first condition to infer that this path also allows $f^i = g^i$ for all $i \geq 1$.

Thus, all directed paths from v allow $f^i = g^i$, for all $i \geq 1$, thereby establishing the equality of descriptors corresponding to v.

□

Getting back to our Example 3, we observe from the Comparison-Graph in Figure 6(a) that the node (q, h) is labeled 'False'. Since this node is accessible from (g, f), we infer that $g \neq f$. However, there is no directed path from any of the other nodes to the 'False' node. Thus, nodes (p, g), (r, q) and (h, f) represent equalities that are true. (The table of equality results is correspondingly updated.)

Thus, we have established that the function descriptor for g is not equal to the function descriptor for f. Therefore we need to add g to our canonical set in order to proceed with the construction of a canonical function descriptor for f. The left leaf node of the LIBDD for f would point to this new member representing g (after we construct it). As for the right leaf which points to h (corresponding to x^k), we already have the result that $f = h$. So the right leaf node points back to itself.

We now move on to constructing the canonical descriptor for g. The left leaf of the LIBDD for g points to p. Since we again have the result $p = g$, this points back to the descriptor for g. Its right leaf points to q, so we need to obtain a canonical descriptor for q. Since the existing canonical set now contains descriptors for f and g, we check q against each using Algorithm A and the associated Comparison-Graphs. Both checks fail at Step 1 on the top-level node itself, implying that we have to add q also to the canonical set. Both leaf nodes of the LIBDD for q are already available in the canonical set (descriptors for f and g). Thus, at

this point, all canonical function descriptors (CFDs) have been constructed, and are as shown in Figure 6(b) (substitutions on pointers from leaf nodes are shown as '$(k-1)/k$').

7. Manipulation of Linearly Inductive Functions

In this section we describe symbolic manipulation of linearly inductive functions, by use of appropriate operations on their Basis BDDs and LIBDDs.

7.1. Boolean Operations

Given two linearly inductive functions f and g, the result h of a Boolean operation on f and g is also a linearly inductive function. With f and g available in a canonical form, obtaining the canonical representation of h is similar to that described in the previous section for arbitrary Boolean combinations of user-specified functions. The only difference is that canonical function descriptors, instead of tentative function descriptors, are used for f and g.

The Basis BDD for h is constructed using a standard BDD *Apply* operation [6] corresponding to the desired Boolean operator on the Basis BDDs for f and g. The resulting Basis BDD is canonical by construction.

The LIBDD for h is constructed using the following steps :

- A tentative LIBDD for h is constructed by a modified Apply operation on the LIBDDs for f and g. This modified LBDD-Apply operation is similar to the standard BDD operation, except that termination of recursion at the leaf nodes may involve creating new functions denoting required (and distinct) Boolean combinations.
- Canonical forms for these new functions and h are maintained using the scheme outlined in the previous section. (In some sense, this step is like a clean-up operation, analogous to performing a Reduce within the top-level BDD Apply after its recursive application to the sub-trees.)

We again use the meta-space to keep track of new functions that denote arbitrary Boolean combinations of given functions. In fact, we use an integrated meta-space to both build the initial set of canonical function descriptors and to add Boolean combinations later as needed. This integrated manipulation is illustrated with the help of the following example.

Example 4 : Let us consider $h = f \wedge g$, where :

$$\begin{array}{ll} \text{for all } i > 1, e^i = (x^i \wedge y^i) \vee ((x^i \oplus y^i) \wedge \epsilon^{i-1}) & \text{and } \epsilon^1 = x^1 \wedge y^1, \\ \text{for all } i > 1, f^i = (x^i \wedge y^i) \vee e^{i-1} & \text{and } f^1 = x^1 \wedge y^1, \\ \text{for all } i > 1, g^i = x^i \vee (y^i \wedge g^{i-1}) & \text{and } g^1 = x^1 \vee y^1. \end{array}$$

We already have the canonical function descriptors for e, f and g as shown in Figure 8(a). We first construct tentative function descriptors for h and a new function p (where $p^i = e^i \wedge g^i$), also shown in the same figure. Note that both h and p correspond to distinct Meta-BDDs, as shown in Figure 8(b). In making h canonical, by checking its equality against the canonical function descriptor for ϵ using Algorithm A, we obtain a Comparison-Graph as shown in Figure 8(c). Since there are no 'False' nodes in the graph, it is easily seen that equalities corresponding to both nodes (h, e) and (p, e) are true. Thus canonical function descriptors for both h and p point to that for e, and no new member needs to be added to the canonical set.

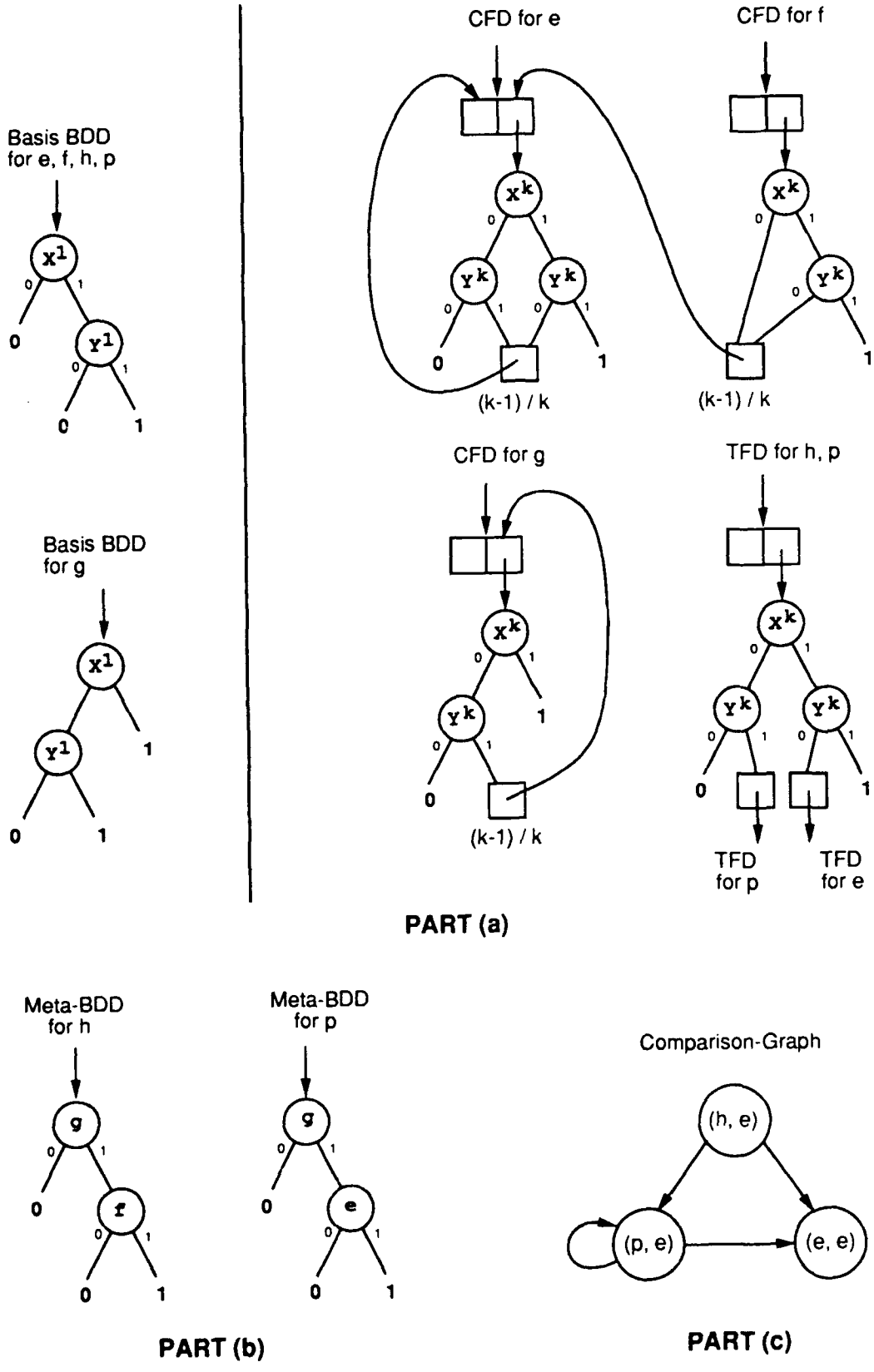


Figure 8: Example 4

7.2. Restriction and Composition

Restriction is defined for an i-instance function f^i with respect to some i-instance variable x^i (denoted $f^i|_{x^i}$), as follows :

- For $i = 1$, restriction of f^1 for $x^1 = 1/0$ is performed by a standard BDD *Restriction* operation [6] applied to the Basis BDD for f.
- For $i > 1$, restriction of f^i for $x^i = 1/0$ is represented, using substitution semantics, as a restriction of the LIBDD for f for $x^k = 1/0$. This operation is also a standard BDD Restriction operation. Note that the leaf nodes of the LIBDD remain unaffected since they implicitly represent other $(k-1)$-instance functions which are independent of x^k. Canonicity of the resulting LIBDD can be maintained by a simple LBDD-Reduce operation, since the leaf nodes of a canonical LIBDD are already available in canonical form.

Composition is defined for the case of an i-instance function f^i composed with another i-instance function g^i, and is useful for modular descriptions of parametric designs. Boolean composition can be expressed in terms of Boolean operations (negation, disjunction, conjunction) and the Restriction operation [6], as follows :

$$g^i|_{x^i=f^i} = f^i \wedge g^i|_{x^i=1} \vee \neg f^i \wedge g^i|_{x^i=0}$$

Thus, a composition operation is performed by combining the Boolean operations and the Restriction operation on the Basis BDDs and the LIBDDs for f and g.

8. Complexity Analysis

We first make the following observations regarding an upper bound on the number of canonical function descriptors :

- Let the number of 1-level variables be m.
  There are no more than 2^{2^m} distinct functions of these variables.
  Thus, there are no more than 2^{2^m} Basis BDDs.
- Let the number of k-level variables be n.
  There are no more than 2^n leaf nodes of a k-level LBDD.
- Thus, the number of distinct 2-level LBDDs is no more than $(2^{2^m})^{2^n} = 2^{2^{n+m}}$.
- Since all functions in the system are linearly inductive, the graph structure of a 2-level LBDD for a function is mirrored for all levels $i > 1$. Therefore, there are as many distinct i-level LBDDs as there are distinct 2-level LBDDs.
- The number of distinct 2-level LBDDs already takes into account the possibilities for the distinct Basis BDDs. Therefore, the number of distinct linearly inductive functions (each identified with a Basis BDD and an LIBDD) is no more than the number of distinct 2-level LBDDs.

Thus, the total number of distinct linearly inductive functions is no more than $2^{2^{n+m}}$, which is also equal to the maximum number of canonical function descriptors.

8.1. Complexity of Canonicity Maintenance

Our method for maintaining canonicity of representations for linearly inductive functions centers around the task of generation and canonicalization of tentative function descriptors. The maximum number of tentative function descriptors is bounded by the number of distinct Boolean combinations of user-specified functions. Thus, if the number of user-specified functions is u, the maximum number of tentative function descriptors is 2^{2^u}.

Recall that Algorithm A is used to check equality of pairs of functions descriptors recursively, leading to construction of a Comparison-Graph for each top-level call. Note from Theorem 2 (Section 6.2) that the number of equality results for pairs of function descriptors, obtained from reachability analysis of a Comparison-Graph, is of the order of the number of nodes in the graph. Since we record these equality results in a table, the maximum number of graph nodes that need to be recursively explored using Algorithm A is bounded by the number of such pairs, which is $(2^{2^u} \times 2^{2^u})$. Thus, number of nodes in a Comparison-Graph is $O(2^{2^{u+1}})$ and number of edges is $O(2^{2^{u+1}} \times 2^n)$ (where n is the number of k-level variables, leading to potentially 2^n dependency edges for each node).

Now, Steps 1, 2 and 3 of Algorithm A, used per node in a Comparison-Graph, are of time complexity $O(2^m)$, $O(2^n)$ and $O(2^n)$ respectively (including computation needed to build the BDDs on their respective variables). As for Step 4, both the depth-first construction of a Comparison-Graph and the subsequent reachability analysis can be accomplished in time linear in the size of the graph. Thus, the total running time to maintain canonicity of all function descriptors is $O(2^{2^{u+1}} \times 2^{max(m,n)})$.

In practice, the number of graph nodes actually explored will be much less than $2^{2^{u+1}}$, since all Boolean combinations of user-specified functions may not be required. Also, by maintaining equivalence classes of function descriptors, we can use the transitivity property of equality of descriptors to avoid an explicit equality check for some pairs. This will cut down the number of times Algorithm A is invoked for node exploration in a Comparison-Graph, thereby improving the actual running time needed to maintain canonicity.

8.2. Comparison with Complexity of Equivalent BDD Manipulations

As described in the previous sections, manipulating linearly inductive functions typically requires simple extensions of the standard BDD operations, followed by maintenance of canonicity of the resulting (and newly created) function descriptors. The contribution due to the BDD-like operations is typically linear in the size of the Basis BDDs and the LIBDDs, which is $O(2^m)$ and $O(2^n)$ respectively (size of a BDD with m/n variables). The dominant term is therefore due to maintenance of canonicity, which is $O(2^{2^{u+1}} \times 2^{max(m,n)})$, as described in the previous section.

Compare this now to BDD representations of linearly inductive functions. It is obvious that BDDs can be used to reason only about particular instances of these functions, not about *all* their instances as our approach can. Thus, our approach is the only alternative if one needs to reason about all instances. However, even for the case of reasoning about a particular i-instance of a function, the complexity of our manipulation operations is independent of i, whereas the complexity of equivalent BDD manipulations would be typically linear in the size of the corresponding BDD, which is linear in i (due to bounded circuit width [2]). This observation provides the justification for using our function descriptor representations (with Basis BDDs and LIBDDs), rather than standard BDD representations, for reasoning about linearly inductive functions.

9. Automating Induction Proofs for Linearly Inductive Functions

The canonicity property of our representation of a linearly inductive function immediately implies that tautology-checking of Boolean formulas constructed from these functions is trivial.

This allows reasoning by induction for these functions as follows. Suppose a certain property is to be proved about each i-instance ($i \geq 1$) of a linearly inductive function f. Also suppose that this property can be represented as a Boolean formula on f. Call this Boolean formula g, and note that g is also linearly inductive. Proving the property for f by induction amounts to tautology-checking of g. In our approach, tautology-checking proceeds by using Algorithm A to check equality of the tentative function descriptor for g and the canonical function descriptor for '1' (which consists of a Basis BDD representing the Boolean constant '1', and an LIBDD which simply points back to the canonical function descriptor). A proof by induction for f translates to a run of Algorithm A on the pair (g, '1') in the following manner :

- Basis Step of the Induction : Check that the basis instance of f (f^1) satisfies this property.

  This translates to checking that g^1 is true, i.e. checking that the Basis BDD for g is equal to the BDD representing the Boolean constant '1'.
  This is done by Step 1 of Algorithm A.

- Induction Hypothesis : Assume that the property is true for some $(k-1)$-instance of f (all i-instances of f, $i < k$).

  This translates to the assumption that the $(k-1)$-level LBDD representing g^{k-1} is equal to the $(k-1)$-level LBDD representing '1' (all i-level LBDDs for g are equal to i-level LBDDs for '1', $i < k$).

- Induction Step : Prove that the property is true for f^k.

  This translates to proving that the k-level LBDD representing g^k is equal to the k-level LBDD representing '1'.

  In our approach, g^k is implicitly represented as the LIBDD for g. Steps 2, 3 and 4 of Algorithm A reason about this LIBDD. Step 4 of this algorithm may recursively lead to equality checks for other pairs of descriptors. Recall that this recursion and the associated equality dependencies between pairs of function descriptors are captured by a Comparison-Graph. For any cycle in this graph accessible from the top-level node (g, '1'), going around the cycle amounts to encountering a new pair which matches an old pair seen before. Note that the new pair implicitly represents a tautology-check for some i-instance of g, where $i < k$. Thus, declaring the top-level node to be true due to presence of cycles (that cannot reach 'False') amounts to using the induction hypothesis on the i-instance of g in order to prove the induction step. In fact, it is this very inductive reasoning that we had used in the proof of Theorem 2 (Section 6.2) to justify our algorithm for checking equality of function descriptors. Thus, the induction hypothesis is taken into account *automatically* by our method for maintaining canonicity of function descriptors.

It is this automatic incorporation of the hypothesis that enables us to completely automate induction proofs for linearly inductive functions, with the canonicity feature of our representations ensuring that no useful hypothesis is ever missed.

References

[1] D. L. Beatty, R. E. Bryant, and C.-J. H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In W. J. Dally, editor, *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, pages 98–112. MIT Press, 1990.

[2] C. L. Berman. Ordered binary decision diagrams and circuit structure. In *Proceedings of the IEEE International Conference on Computer Design*, pages 392–395, 1989.

[3] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic simulation and temporal logic. In *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 759–764, Leuven, Belgium, 1989.

[4] S. Bose and A. L. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *Proceedings of the IEEE International Conference on Computer Design*, 1989.

[5] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] R. E. Bryant. A methodology for hardware verification based on logic simulation. Technical Report CMU-CS-87-128, Computer Science Department, Carnegie Mellon University, June 1987.

[8] R. E. Bryant and C.-J. H. Seger. Formal verification of digital circuits using symbolic ternary system models. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1991.

[9] J. Burch, E. M. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society Press, June 1990.

[10] J. Burch, E. M. Clarke, K. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, June 1990.

[11] A. J. Camilleri, M. J. C. Gordon, and T. F. Melham. Hardware verification using higher-order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 43–67. North-Holland, Amsterdam, 1987.

[12] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model-checking algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 294–303, August 1986.

[13] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, 1989.

[14] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[15] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1991.

[16] S. M. German and Y. Wang. Formal verification of parameterized hardware designs. In *Proceedings of the IEEE International Conference on Computer Design*, pages 549–552, Port Chester, NY, 1985.

[17] W. A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–460, 1989.

[18] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 239–247, Edmonton, Alberta, 1989.

[19] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, 1991.

[20] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80, Grenoble, France, June 1989. Springer-Verlag.